A COMBINED LOGICAL AND FUNCTIONAL PROGRAMMING LANGUAGE

Michael O. Newton

Computer Science
California Institute of Technology

5172:TR:85

A COMBINED LOGICAL AND FUNCTIONAL PROGRAMMING LANGUAGE † ‡

Michael O. Newton
California Institute of Technology
Pasadena, CA. 91125

Caltech CS Technical Report 5172:TR:85

Abstract

Since the development of Prolog in 1974, there has been increasing interest in the area of logic programming. Though Prolog is the most often used language used for logic programming, it still is lacking many features found in traditional languages such as LISP. These features include functional notation, array handling, and the concept of modularization.

This thesis describes a method of adding a functional style of programming to Prolog. The addition has been a conservative extension – the language is completely compatible with existing Prolog programs. In addition to a working prototype, a method of compilation is presented.

---

† This research partially supported by a contract from the IBM Corporation.

‡ The research described in this report was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

Contents

Chapter 1: Introduction

When computers were first developed, programmers wrote their instructions to the machine in machine languages. Being extremely tedious to program, machine languages were quickly replaced with the development of assembly languages. These relieved the programmer from tedious address calculations, and allowed the use of symbolic names for locations in memory. But, there were as many machine languages as there were different types of machines, and, out of necessity, each assembler language reflected its own particular machine language.

In the 1950's Grace Hopper and others developed the idea of a programming language – these were relatively portable from machine to machine, saved the programmer from tedious byte and bit counting, and had as input programs that were much clearer and easier to maintain than assembler or machine language. Two of the languages developed during this time were Fortran and Lisp.

For many years Lisp stood apart from all other commonly used languages. It was designed as a language for dealing with lists and nested levels of data. Unlike the others, (pure) Lisp worked without state or so called side effects. Thus, Lisp was the first functional language – the primitive operations were evaluation and quoting (prevention of evaluation). One could pass functions as arguments to other functions and one could return functions as a results.

In part, the success of Lisp was also due to its underlying simplicity: There was (at first) but one data structure – the list. An array could be a list of rows, each of the rows being another list of elements. A string was a list of characters. Even the Lisp program was a list – There was no distinction between the program and the data.

Among certain computer science subcultures, most noticeably the Artificial Intelligence community, Lisp became 'the' programming language. The various design and implementation decisions were all examined in depth: garbage collection, efficient handling of arrays, Input/Output (and its associated state). Lisp compilers were developed to speed up a rather slow interpretation mechanism. It became a very robust programming language.

In 1974, more than twenty years after the development of Lisp, the work of Colmerauer, Van Emden and Kowalski generalized the mathematical idea of functionality one step further and designed a relational language: Prolog. For several years it remained a rather obscure language used almost exclusively in Europe. However, by 1980 it became noticed by researchers in America – most noticeably by the Artificial Intelligence community. Later, the Japanese Fifth Generation Project (ICOT) picked it as the base language for their proposed new supercomputers. Interest spread quickly.

Since the mathematical concept of a relation is a superset of a function, Prolog can be considered at least as powerful as pure Lisp. In addition, Prolog includes backtracking – if at any time in the computation trail a goal fails, (see chapter 2 for a brief Prolog introduction), previously executed goals (and their variable bindings) are retried. Thus, unlike most languages, Prolog programs tend to have only minor amounts of control statements – no littering of `for`'s, `goto`'s, `do`'s, `while`'s, or `if`'s – Prolog automatically searches through the solution space for the programmer. Due to this, Prolog programs can be thought of as a specification of the answer space, and not as a sequence of instructions describing how to find a particular answer.

Unlike the very mature Lisp, Prolog is new and suffers the problems of most new languages. All the "bells and whistles" that make a programming language much easier to use, and that have been incorporated into a well known language, are missing. Certain often repeated constructs are cumbersome, or lacking.

In Chapter 2, some of these problems will be introduced, along with a brief descriptions of a possible solution and a review of the work of other researchers in this field. Chapter 3 will introduce Prolog.

Chapter 4 will describe the new language and provide some programming examples that show the usefulness of the new language. Chapter 5 will go into the (behind the scenes) details of the design new language AT1, and, Chapter 6 will hopefully offer insight into many of the particular implementation decisions that were made.

Though there is a working prototype, it runs extremely slowly and uses vast amounts of storage due to the two levels of interpretation that it uses. Chapter 6 will example just how efficient the current prototype is. Conversely, and of more practical use, Chapter 7 will explain the mechanism by which AT1 code can be compiled – one of the more exciting results.

Finally, Chapter 8 will be a bull session where the author states his opinions on what other extensions are necessary to make Prolog/AT1 into a more useful language. The two appendices contain the full listing of the interpreter and a large sample program.

Chapter 2: An Overview of the Problem and a Proposed Solution

This chapter will present some of the features that the author believes are lacking in current implementation of Prolog, along with a brief description of what the author feels is a more useful solution. In addition a brief description of the current prototype is discussed.

Conventions

Throughout this and following chapters, Peano notation will often be used when dealing with Prolog code. In this notation 0 represents the ordinal zero and s(X) denotes the successor to X. Thus, s(s(s(0))) represents the ordinal 3.

Lisp code presented in the following pages was run under Franz Lisp.

The Problems

In most conventional programming languages, it is very easy to add two numbers – in Fortran the statement

```
J = 3 + 2
```

assigns to the variable J the value 5 – a process which changes the state of the variable, and thus changes the state of the computation. This is done so that the value in J can be used in further computation.

In Lisp, it is also rather easy to add integers – If integers exist as primitives in the current system, then there is usually a primitive to add them together. As an example, in Franz Lisp, one could type

```
(add 3 2)
```

and the interpreter responds with a 5. In pure Lisp, no state needs to be preserved. In order to use the result, one uses functional composition:

```
(mul (add 3 2) (add 1 8))
```

If the version of Lisp does not have integers as primitive objects, it is still possible to define add using the successor function:

```
(defun adds (x y)
        ( cond  ( (eq x 0) y)
                ( (eq (car x) 's) (adds (cadr x) (list 's y)) )
                ( t (msg "Error"))
        )
)
(defun muls (x y)
       ( cond   ( (equal x 0) 0)
                ( (equal (car x) 's) (adds (muls (cadr x) y) y) )
                ( t (msg "Error"))
        )
)
```

Using these definitions, we can type:

```
        (muls '(s (s (s 0))) '(s (s (s (s 0)))) )
```

and get:

```
        (s (s (s (s (s (s (s (s (s (s (s (s 0))))))))))))
```

In Prolog it is equally easy to define addition and multiplication using successor functions:

```
add(0,Y,Y).                                     /* 0 added to Y is Y */
add(s(X),Y,s(R)) :- add(X,Y,R).                 /* (1+X) + Y = 1+R if R=X+Y */

mul(0,Y,0).                                     /* 0 times Y is 0 */
mul(s(X),Y,R) :-  add(Temp,Y,R), mul(X,Y,Temp).
                                                /* (1+X)*Y=R if X*Y=T and T+Y=R */
```

Notice that while the Lisp version had 2 arguments to each function, Prolog has 3 arguments to each *relation*. This represents one of the most significant differences between Prolog and most ordinary languages, for one can not only request the following goal:

```
        mul( s(s(s(0))) , s(s(0)) , Result ).
```

But also, the goal:

```
mul( F1, F2, s(s(s(s(s(0))))) ).
```

and Prolog will dutifully print out:

```
F1 = s(0)
F2 = s(s(s(s(s(0))))) ;
F1 = s(s(0))
F2 = s(s(s(0))) ;
F1 = s(s(s(0)))
F2 = s(s(0)) ;
F1 = s(s(s(s(s(0)))))
F2 = s(0) ;
```

or, either of the following goals:

```
mul(F1, s(s(0)), s(s(s(s(s(0))))) ).
mul(F1, F2, M).
```

for which Prolog will print out all legal combinations.

This power is always there – even when all we want are simple functions as opposed to relations. In addition, not only must the necessary computational overhead be there, we must write everything that would normally be a function of N arguments as a relation of N+1 arguments. The disadvantage of this is shown by the following example, first in normal Prolog:

```
add(3,A,Temp1), add(B,C,Temp2), mul(Temp1,Temp2,Temp3),
add(A,C,Temp4), mul(D,Temp4,Temp5), add(Temp3,Temp5, Result).
```

Then, with functional notation:

```
add( mul(add(3,A) , mul(B,C))    ,   mul(D , add(A,C)) ).
```

And, finally, using the infix notation of Prolog:

```
Result = ( (3 add A) mul (B add C) ) add ( D mul (A add C) ).
```

Adding a functional notation to Prolog requires the design of both the semantics and syntax of the new rules, along with the interface between the two logics. The commentary in Chapter 5 discusses many of the more difficult design decisions in detail.

In essence, the major part of the work for this thesis was the clean addition to Prolog of functional notation. To accomplish this, a new form of database rule was added to Prolog to allow a user to specify rewrite rules. These rules were allowed to depend on Prolog clauses. Then the user could switch back and forth between functional clauses and relational clauses.

To do this, rules of the form:

```
a => b :- c.
```

are added to the database to specify rewrite rules. The above clause specifies that if we are currently rewriting a term that unifies with 'a', then we can rewrite it to 'b' if the Prolog goal(s) 'c' are true.

Other work

The goal of this research – to form some fusion between the logic programming style of Prolog and a functional/rewriting style – has been attempted by several others. Most of the proposals attempt to model term equality by invoking rewrite systems. Most have proposed using restricted forms of full paramodulation. Although known to be complete, full paramodulation has several disadvantages. First, it is very inefficient. Paramodulation proof trees exhibit huge branching ratios that overwhelm most straightforward search paradigms. Second, as opposed to theorem proving, the semantics of paramodulation in a logic programming environment offers less than clear semantics. In particular, there is a fundamental shift in the way Prolog terms must be viewed: at any given time, a term may actually be a data structure or a function call – in highly unpredictable ways. In some sense, many of the proposals which attempt to model paramodulation suffer from these same defects.

The first major attempt was done by J.A. Robinson and E. E. Sibert – the design of Loglisp. Loglisp is an extension to Lisp provided by special functions that invoke logical deduction. In particular, `ALL`, `ANY`, `THE`, and `SETOF` are added to Lisp and are executed during the deduction step of Loglisp's evaluator. Another similar paper is Mats Carlsson's work in describing the procedural semantics of Prolog using functional programming constructs.

Another such system is Poplog. Poplog was designed to look very much like standard Prolog, but with an interface to other languages – Pop-11, Fortran, or others. Each language can invoke the other, and data can be passed between them.

Funlog, by P. Subrahmanyan and J-H. You, modifies the unification step of Prolog to accept "semantically unifiable" terms. In addition they offer a form of lazy evaluation. Barbuti, Bellia, Levi and Martelli's work also offers a combined functional and logical programming language.

Tamaki's work is perhaps the closest to ours in spirit. Tamaki introduces a rewrite predicate which, in contrast to paramodulation, may be conditioned by other goals. He translates restricted forms of rewrite rules into Prolog clauses, allowing nondeterministic rewrite behavior.

The author would like to point out the work done by Burstall and Goguen, Goguen and Meseguer, and by Futatsugi, Goguen, Jouannaud and Meseguer in designing functional and logic programming languages that contain theories, types and modules. The reader is encouraged to consult their work.

The similarity between AT1 and Prolog suggests that a direct translation of AT1 into Prolog might be possible. Unfortunately, this is not the case. Due to the vagaries of variable binding, the

interpreter would have to be resident and frequently invoked. A direct compilation of AT1 into Prolog machine instructions allows the interpreter to call compiled code, thus reducing the amount of interpretation needed.

Chapter 3: Background

This chapter serves as an introduction to the notation and terminology used throughout this report. It also presents an overview of current standard Prolog (In reality there is no such beast.).

Prolog Syntax, Semantics, and Execution

The syntax of Prolog is simple: clauses ending with a period. In the simplest form these would look like:

```
clause1.
```

A more detailed description of a clause is a functor and some nonnegative number of arguments. If the number of arguments is non-zero, these arguments appear in parenthesis following the functor. Note that in current implementations there cannot be a space between the functor and the opening parenthesis. Here are some more examples:

```
a_functor( an_arg, another_arg, junky_arg ).
foaming_frenzy.
```

Notice that so far all terms have had lower case initial letters. These indicate atoms. A variable is any word that starts with an uppercase letter.

In a Prolog term the outermost functor is called the principal functor. There are several special principal functors. ":-" is the most special. Whereas all the above clauses, when inputed to the interpreter would be interpreted as facts, (or queries if typed in query mode), clauses with the principal functor :- are treated as conditional statements. These can be considered as if statements – the left hand side is true if (*not iff*) the right hand side is true. An example:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,T), parent(T,Y).
```

This example defines ancestor of X to be Y if Y is the parent of X, or if there exists T such that T is an ancestor of X and the parent of T is Y. Notice the implicit 'or' implied by the pair of statements, and the explicit 'and' denoted by ','. This definition could also be written as:

```
ancestor(X,Y) :- parent(X,Y) ;
                 ancestor(X,T), parent(T,Y).
```

Using the explicit or ';'. (The fact that these two statements are spread across lines is for the reader. Prolog pays no attention to this).

Several other observations: Variables are local to a clause – an X in one clause has no relation to an X in any other clause. Also, variables that appear on the left side can be considered universally quantified, and variables on the right side can be considered existentially quantified. Thus the above example would be read: "For any X, Y is the ancestor of X if either Y is a parent of X or, if there exists a T such that Y is a parent of T and T is an ancestor of X."

Notice, also, that a fact is represented by:

```
fact.
```

when the database manager is active or as:

```
assert(fact).
```

when the prover is active.

Conversely, a query is of the form:

```
:- query.
```

when talking to the database manager.

When discussing Prolog clauses it is customary to refer to them by their name and arity, separated by a slash. Thus a procedure with head "in(A,B)" would be referred to as in/2. Functions of different arity are completely independent.

The semantics of Prolog can be considered in two different ways: Procedurally and Declaratively. Declaratively the program (facts and conditional statements) are considered as a declaration of the problem, and it is up to Prolog to find the answer(s) given the known constraints. Prolog will do this by searching through the allowable search space, keeping track of all decisions points, until either an answer is found, proof of the impossibility of the problem is proved, or, in the worse case, the specification causes an infinite loop in the prover.

A Prolog program can also be viewed as a procedural specification of how to solve a given task. In this mode, the head of a clause is the procedure that is called, and the statments that form the body of the procedure are the statements to be executed in order to "return" from the procedure.

For a detailed formal description of the theorom proving techniques used in Prolog, the reader is refered to Robinson's 1965 paper.

Chapter 4: An Informal Description of AT1

AT1 is a conservative extension of Prolog. That is, unmodified Prolog programs will compile and execute under AT1 as in (DEC-20) Prolog (Pereira, et. al. 1978). AT1 programs consist of two sets statements: A set of logical clauses, as in Prolog, as well as as set of conditional rewrite rules. These statements may appear in any order. Only the order within the two sets of clauses is important, and, even then, only for the operational semantics of Prolog. The logical clauses are identical to Prolog in syntax and semantics. A sample conditional rewrite rule looks like this:

```
% Generate an ascending list of integers from M to N, inclusive.
iota(M,N) => [M|=>iota(M+1,N)] :-  M<N.
iota(N,N)=> [N].
```

The first rewrite rule is *conditioned* by the clause `M<N`. So that in order to use this rewrite rule a Prolog subgoal must first be solved. Of course, several clauses may appear after the `:-` representing a goal list whose conjunction must be first solved before applying the rewrite rule. Note that unification is used in the second rewrite rule to ensure that the two arguments are equal. Finally, note the presence of the *monadic rewrite symbol* (written as `=>`) in the first rule that forces rewriting of the newly rewritten term. We prefer to read this as "The value of `iota(M,N)` is the list with the head `M`, and the tail whose value is `iota(M+1,N)` as long as `M<N`".

Informal Explanation: Examples and Explanation

We now give several examples. Their purpose is twofold. First, we wish to familiarize the reader with the AT1 programming style. Second, we wish to present code samples which illustrate particular difficulties in compilation. We later show how to overcome these problems.

Example 1. The quintessential Prolog example is list concatenation. Below, we present both the standard Prolog definition, and the AT1 version of append as a function.

```
% An append relation defined by logical assertions.
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
append([],L,L).

% An append function defined by rewrite rules.
append([X|L1],L2) => [X|=>append(L1,L2)].
append([],L) => L.
```

Note that both examples have a natural declarative reading. The first example is the celebrated predicate calculus reading. The second example reads left to right as follows. "The appending of a list with head X and tail L1 with list L2 is the list with head X and tail given by the value of appending L1 with L2."

Example 2. A primes sieve.

```
% Generate a list of primes up to N.
primes(N) => =>sieve([],iota(2,N)).

% Sieve the primes list against the list of remaining integers.
sieve(PList,[Prime|NList]) =>
                    =>sieve([Prime|PList],del_mul(Prime,NList)).
sieve(PList,[]) => =>reverse(PList).  % Done, reverse the list.

% Delete multiples of N in L.
del_mul(N,L) => =>del_mul(N,append(H,TP)) :- append(H,[HP|TP],L),
                                             divides(N,HP).
del_mul(N,L) => L.

% divides(N,HP) :- integer(=>HP/N), HP is (HP/N)*N.
divides(N,HP) :- N=1; N=HP.
divides(N,HP) :- HP>N, divides(N,=>(HP-N)).
```

In this example, we use the Prolog clause `divides` to determine if the first argument is a divisor of the second argument. In doing so, the monadic rewrite symbol tells Prolog that the term `HP/N` should be replaced with its value, found by rewriting it. This interpetation coincides with the that for the monadic rewrite symbol when encountered during rewriting (as in the `iota` example).

We then use the `divides` predicate in the function `del_mul`. `del_mul(N,L)` is the sublist of L with integer multiples of N removed. In `del_mul`, the two appends serve to pick out and then delete the element HP from the list L. This element is guaranteed to be a multiple of N by the `divides` clause. Finally, if there are no more multiples of N in L, `del_mul(N,L)` is defined to be the list L.

The definition of `sieve` states that after deleting all multiples of a prime in L, the lowest element remaining in L is the next prime. Finally, if there are no more elements left, `sieve` is the sorted prime list.

The next three examples demonstrate some of the possible ways that Prolog clauses can interact with the rewrite rules. These illustrate features of AT1 that make compilation difficult – and will be discussed again later in this paper.

Example 3. Quoting, a simple example.

```
% Illustrations of quoting
foo() => 1+1.
glue() => =>(1+1).

bar1 :-  Y=1, (#X) = #(=>1+Y),is_two(X).
bar2 :- (#X) = #(=>1+Y), Y=1, is_two(X).

is_two(2).
```

The rewrite rule `foo` will rewrite to the term `(+(1,1))`. Note that the Prolog term is returned without any further evaluation. In `glue()` further evaluation is explicitly invoked by the monadic rewrite symbol.

In bar1 the variable X becomes bound to =>(+(1,Y)), however, since quasi-quotation does not prevent variable binding, and Y is already bound to 1, X becomes bound to =>(+(1,1)). In bar2 this occurs after the execution of the second subgoal. Finally, in either case, the call to is_two forces the evaluation of the input argument X, and both bar1 and bar2 succeed.

Example 4. The order of clauses is important, thus implying the incompleteness of deduction with rewriting.

```
g :- X = =>(1+Y), Y=1, is_two(X).
h :- Y=1, X = =>(1+Y), is_two(X).

j :- X = =>(1+Y), is_two(X), Y=1.
k :- Y=1, is_two(X), X = =>(1+Y).

is_two(2).
```

In this example the order of subgoals influences the refutability of the two clauses. The clauses, g, h, and k succeed, while the clause j fails. The execution of clauses g and h proceeds the same as in the previous example. The execution of the subgoal is_two(X) in j forces the evaluation of an expression without a value, and thus cannot be rewritten. Thus during the unification in procedure is_two we encounter failure in attempting to rewrite the term 1+Y. In k, is_two binds X to 2, so during the execution of X= =>(1+Y) the unifier will force the evaluation of (1+Y) which equals (1+1). Thus k will succeed.

Example 5 Principal functor supplied by rewriting

```
f(X) :- => X.
```

This short example shows how rewriting can provide to Prolog the term it will use for a subgoal, thus forcing invocation of the Prolog interpreter.

Example 6 Infinite data structures:

```
fib(N) => =>upto(N,f(1,1)).

f(A,B) => [A|f(B,A+B)].

upto(0,X) => [].
upto(1,[H|T]) => [H].
upto(N,[H|T]) => [H|=>upto(N-1,T)] :- integer(N).
```

Here, fib(N) produces the first N Fibonacci numbers using an algorithm shown to the author by Alain Martin. To do this, it calls the predicate upto(N,F) which will evaluate only the first N elements of the list generated by the function F. In this case f/2 is a function defined in terms of itself, and, if it were to be evaluated directly, would take an infinite amount of time.

In appendix A, we present the source of a large AT1 program. While it does not run efficently under the current prototype system, this prototype runs under both compiled DEC-20 Prolog and under

C-Prolog. Aproximately 40% of the code are utility functions like `union`, `diff`, `append`, `ranunif`, (the generator of a random number on a given interval), `+`, `-`, and `*`, all of which are nothing more than the functional equivalent of the often used Prolog predicates with the same names. Another large percentage (aproximately 25%) of the code prints the internal representation of the maze in a form for human solution. This code takes an inordinate amount of the time in the prototype AT1 because it involves long lists of characters that must be rewritten. Finally, the main code to make a maze consists of the goals `make_maze`, `add_seg`, `ok`, `rand_pt`, and `all_segs`.

The goal `make_maze(H,W)` makes a maze of size $H*W$ by adding $(H-1)*(W-1)$ randomly placed segments segments from the list `all_segs(H,W)` using the procedure `add_segments`. `add_segments` generates random neighbor pairs of points in the region within the maze, and then uses append to make sure this point has not already been chosen. Finally, `ok` is called to make sure that the maze is still solvable, and that there are no closed boxes. These parameters are checked for by checking whether there is a path from one side of the new segment added to the other side of the new segment.

A Prolog Operational Semantics for AT1

We now present the core of an AT1 interpreter written in Prolog. Note that this program does not execute the same under AT1 as under Prolog since it makes use of rewrite symbols and quotes. Note that all system predicates (`print`, `atom`, `var`, ...) must be modified so that unification occurs in a consistent manner. In addition, the interpreter does not handle the Prolog 'cut' operator. Appendix B contains a complete listing of the working interpreter.

First the basic Prolog interpreter, modified so we can get at unification:

```
% The basic goal:
prove(true).
prove( (A,B) ) :- prove(A), prove(B).
prove( =>A ) :- eval(A,X), prove(X).
% special code for system predicates goes here
prove(A) :- our_clause(A,B), prove(B).
```

This is nothing more than the standard Prolog interpreter in Prolog (Pereira, et.al. 1978), with the addition of one clause to handle principal functors generated by rewriting.

Next, clause has been changed to call our unifier:

```
% get a candidate clause from the database.
our_clause(A,Q) :- var(A), print("Cant prove var.  Bye"), !, fail.
our_clause(A,Q) :-
                functor(A,Functor,Arity),
                functor(P,Functor,Arity),
                clause(P,Q),
                our_unify(A,P).
```

The unifier has the calls to properly handle a monadic rewrite found during unification:

```
our_unify(X,Y) :- (var(X); var(Y)), !, X=Y.        % Should be first
our_unify(=>X, Proc) :- nonvar(Proc), !, our_eval(X,Y), our_unify(Y,Proc).
our_unify(=>X, Proc) :- var(Proc), !, Proc = (=>X).
our_unify(Proc, =>X) :- var(Proc), !, Proc = (=>X).
our_unify(Proc, =>X) :- nonvar(Proc), !, our_eval(X,Y), our_unify(Y,Proc).
our_unify([],[]) :- !.
our_unify([H|T], [H1|T1]) :- !, our_unify(H,H1), our_unify(T,T1).
our_unify(#X,#X) :- !.
our_unify(X, Proc) :-
                X   =..  [Functor|Xargs],
                Proc =.. [Functor|Pargs],
                our_unify(Xargs,Pargs).
```

The first unify clause insures that variables do not cause rewritting. The next four clauses are the link between Prolog and rewriting. The monadic rewrite symbol causes the value of the term `X` to unified with the procedure head. The sixth and seventh lines simply map `our_unify` to lists. The next line guarantees that quoted terms will not be rewritten. The final clause matches functors and unifies subterms to recursively finish the definition.

Finally, the two parts of the evaluator: `our_eval` handles the original evaluation of a clause.

```
our_eval(X,X) :- (integer(X); atom(X); var(X)), !.
our_eval(#X,X) :- !.
our_eval(=>X,Y) :- !, our_eval(X,Y).
our_eval([],[]) :- !.
our_eval([H1|T1], [H2|T2]) :- !, our_eval(H1,H2), our_eval(T1,T2).
our_eval(X,Z) :- !,
                 functor(X,Functor,Arity),
                 functor(Y,Functor,Arity),      % Make a dummy of same form
                 X =.. [Functor|Xargs],         % Take care of args first
                 our_eval(Xargs,XXargs),
                 XX =.. [Functor|XXargs],       % Build new ars
                 clause( =>(Y,T) , Q ),         % Get an
                 our_unify(XX,Y),               %        appropriate clause
                 our_prove(Q),
                 our_ceval(T,Z).                % See if more work to do?
```

The second part of the evaluator `our_ceval` is called by `our_eval` if, after rewriting, there remains

monadic rewrites. If so, our_ceval will find them, and start the rewriting over again.

```
our_ceval(X,X) :- (integer(X); atom(X); var(X)), !.
our_ceval(#X,#X) :- !.                          %investigate no further
our_ceval(=>X,Z) :- !, our_eval(X,Z).           % Yes there is.
our_ceval([],[]) :- !.
our_ceval([H1|T1],[H2|T2]) :- !, our_ceval(H1,H2), our_ceval(T1,T2).
our_ceval(X,Y) :- !,
                  X =.. [Functor|Args],
                  our_ceval(Args,Nargs),
                  Y =.. [Functor|Nargs].
```

In our_eval a quoted term, #X, becomes X. A monadic rewrite symbol encountered at this point continues with the rewriting – essentially a null operation. An atom, variable or integer evaluates to itself. For convenience the value of a list is the list consisting of the value of it's elements ‡. Finally, we attempt to apply a conditional rewrite rule. We do this by rewriting the subterms, matching against a rewrite rule in the database, and proving the conditions attached to the rule. Afterwards, our_ceval checks for monadic rewrites still present in the goal. Note that at no time must the evaluator bind something to a rewrite symbol.

A Formal Semantics of AT1

The declarative semantics of AT1 proceeds from the declarative semantics of Prolog with two extensions. We first introduce a rewrite predicate of two arguments. We write this with an infix operator, e.g. $t{=>}u$.† We then define a new inference rule for formulas which contain terms with the *monadic rewrite symbol.* Third, we define evaluation by rewriting. Note that we could have foregone an inference rule in favor of adding a rewrite axiom scheme.

We introduce some notation. We will often write a term p which contains a term t as $p[t]$. When we do, we will mean by $p[u]$ the term with that occurrence of t replaced by the term u. If p contains more than one occurrence of t we replace *only one* of the occurrences of t — we note that this is at variance with a similar notation common in mathematical logic.

We define two metalogical predicates: p ***is provable*** and t ***evaluates to*** u written $\vdash p$ and $t \triangleright u$, respectively. In the rules that follow we define these relations for ground terms only. To extend the notion to all Prolog terms we use the convention that the three relations hold over terms if and only if they hold for all ground instances of those terms.

We first define the provability meta predicate on atomic formulas.

1. Axioms. All ground instances of axioms are provable.

2. Horn Clause Resolution. If $p :- q_1, \ldots, q_n$ and $q_1, \ldots q_n$ are provable then so is p.

$$\vdash p :- q_1, \ldots q_n \wedge \vdash (q_1 \wedge \cdots \wedge q_n) \ \textit{implies} \ \vdash p.$$

3. Rewrite Inference Rule. This rule connects the above semantics of Prolog with the rewrite semantics of AT1. If t evaluates to u and $p[u]$ is provable then $p[{=>}t]$ is provable.

$$t \triangleright u \wedge \vdash p[u] \ \textit{implies} \ \vdash p[{=>}t].$$

‡ This is not strictly necessary as the user can enter a clause in his database to do the same thing.

† We shall not allow atoms to sit in the first argument of a rewrite predicate. Rather, we shall introduce *niladic terms* such as `f()` which will be strictly distinguished from atoms.

We now define the evaluate relation between terms.

4. Quoted terms. The quote of a term t, written $\#t$. Evaluates to t.

$$\#t \triangleright t$$

5. Atoms. An atom a evaluates to itself.

$$a \triangleright a$$

.

6. Connection to logic. If the rewrite predicate $f(s_1, s_2, \ldots, s_n) => s_0$ is provable and $t_1 \triangleright s_1, t_2 \triangleright s_2, \ldots, t_n \triangleright s_n$ then $f(t_1, \ldots, t_n)$ rewrites to s_0.

$$t_1{=>}s_1 \wedge t_2{=>}s_2 \wedge \cdots \wedge t_n{=>}s_n \wedge \vdash f(s_1, \ldots, s_n){=>}s_0 \textit{ implies } f(t_1, \ldots, t_n) \triangleright s_0.$$

Implications of the Semantics

Conservation theorem: Let T be a set of clauses, i.e. *a theory*. If T is expressed in the language of pure Prolog, and AT1 refutes T, then Prolog will also refute T.

$$T \in L(Prolog) \wedge AT1 \nvdash T \textit{ implies } Prolog \nvdash T$$

The point of the above theorem is that, the meaning of Prolog programs is preserved. This is in sharp contrast with other proposals to extend Prolog through rewrite rules. In previous schemes the inclusion of a rewrite rule may cause term rewriting at virtually any time, causing unfortunate interactions which modify the meaning of a program significantly. In our scheme, the monadic rewrite symbol (`=>`) strictly controls on what terms evaluation occurs. Thus Prolog programs which do not mention of the rewrite functor will behave exactly the same in Prolog and AT1.

Note that before a rewrite predicate participates in evaluation of a term t, all subterms of t must first be evaluated. Thus, partial evaluations of a term are never values that participate in deduction. For example, the goal `f(=>(1+1+1))` will never resolve with `f(2+1)`. Similarly, the semantics *do not* specify that two identical terms will be rewritten to the same value. For example, if `X` is bound to `blech(uck)` and `blech(uck)` can rewrite to `morning` or to `broken_car`, then the goal `rg(X,X)` can call `rg(morning, broken_car)`.

Notice that the monadic rewrite rule is strictly less powerful than the paramodulation rule in that `f(=>2)` will never resolve with `f(1+1)` *.

Also, note that the rewrite rule is only applicable where it is explicitly invoked via the monadic rewrite symbol. That is, the monadic rewrite symbol is an explicit call to begin evaluation from inside of Prolog. In this way, the large branching ratios caused by coincidental paramodulants are effectively controlled †.

---

* The user is prohibited from entering `2 => 1+1` into the database because this rewrite predicate has an atom in the first argument.

† The quote prohibits further rewriting of a term once encountered. As in Lisp, it is stripped off by evaluation, so that if the term is once again evaluated, it may be further reduced.

Chapter 5: Design Decisions

Many different versions of the proposed language were developed. Each of these suffered some flaw that was 'corrected' in a later version. This chapter covers many of these decisions, partly in justification of the current design. Since Prolog was desired unchanged, most of the decisions dealt with the interface to Prolog and with the semantics of rewriting. In retrospect, the hardest decisions were what to design in the first place.

The Database

The syntactic form of the rewrite rules, though never a major issue, nonetheless reflected other major design decisions. For example, during the first few months rewriting and Prolog theorem proving were considered as much less tightly coupled than in the current version. Thus, in early models, Prolog horn clauses were separated from the rewrite rules – they were even thought of as occupying separate data bases.

Initially rewrite rules looked like (with explicit grouping):

```
f(X) => (g(X-1) if X>1).
```

in the final version, the rules looked like:

```
(f(X) => g(X)) :- X>1.
```

Though this change is small, the change in philosophy is major: In the first line the only major relation to Prolog is the goal `X>1`. The second version is a normal Prolog goal with head `=>(f(X),g(X))`.

Initiating Evaluation

In the early versions of AT1, rewriting was started by the appropriate goal in a Prolog clause, and continued until a quote was found. This deviates far from a natural semantics, since each phase of rewriting must end in failure, or, alternately, rewriting must be reflexive: a term could rewrite to itself. Neither of these choices was appealing.

In the current version, there exists an explicit symbol `=>` to enable rewriting. Since rewriting acts as a form of term replacement – the `=>X` term in a Prolog goal is replaced with a new term – it is no longer the case that rewriting recursively occurs on the Prolog goal. Instead, the `=>/1` functor can be used within a rewrite database clause to express continued evaluation.

Thus, in the first design, the goal `h(2) = =>f(X)` with the database:

```
f(X) => g(X).
g(X) => h(X).

h(2).
g(1).
```

would return true, with X bound to two. This repeated rewriting was very hard to control, even with quoting, since the quotes were just stripped during the next attempt at rewriting. But, by allowing the monadic rewrite within a rewrite rule, and by eliminating the repeated rewritings, the above goal would fail and the goal `g(1) = =>f(X)` would succeed. Or by modifying the database to read:

```
f(X) => =>g(X).

g(X) => h(X).
h(2).
g(1).
```

the goal `h(2) = =>f(X)` would succeed with X bound to two.

There were numerous advantages to this method. It gave the AT1 programmer more control over his or her data. It drastically reduced the number of quasi-quotes necessary in a program, and it saved the language from attempting many rewrites which would not be needed. In addition, coupled with a few other changes – niladic functions and the new order of evaluations – a more logical semantics was possible.

Terminating Evaluation

Complicating the issue of initiating evaluation, was the issue of when to quit rewriting. This could take three forms: a quote, a failure, or completion. First we shall look at quoting. Some of the possible design decision regarding quoting are: 1: What to do when quotes are found while not evaluating, 2: What to do when quotes are found while evaluating, 3: Should quotes match Prolog terms? and, 4: Should quotes be stripped when found (while evaluating?, while not evaluating?).

Believing that it was necessary that any language be able to quote all strings in that same language, it was decided that quoting should be a more powerful operator than evaluation. In particular, the sequence =>x with the data base

```
x => #(=>(f())).
f() => g.
```

returns `=>(f()).`

This demonstrates the stripping of a quote when encountered while rewriting – allowing the return of a structure with a head of either => or #. Inside Prolog (when not rewriting) a # is treated the

same as any other head of a term, thus allowing Prolog to AT1 terms. For instance, suppose we have X bound to =>f(g) and we call the goal p(#X). Then, we can use Prolog to pass X's binding to the database:

```
p(#(=>(Z))) :- print(Z).
f(g) => hi.
```

with the goal:

```
X = =>f(g), p(#X).
```

and AT1 will print f(g) and not hi.

Besides quoting, evaluation can be ended in other ways. In particular, an atom, integer or variable rewrites to itself. The null list is also treated as an atom – a special case of the fact that the value of a list is the list consisting of the values of its elements. In all of these cases rewriting to itself was an optional decision for user convenience – in each case the same effect could be had by adding the appropriate clause(s) to the data base.

A final method of terminating rewriting is failure. Consider the Prolog goal =>f(u). There should then exist a clause in the rewrite data base which will match f(u). Should this not be the case, unification with the term would fail.

To keep the semantics consistent, niladic functions were introduced. Otherwise, the rewriting of atoms to themselves conflicts with the failure that is produced if a term is attempted to be rewritten and there is no applicable clause. Thus, the goal =>f will always succeed and return f. But, the goal =>f() will only succeed if there is a rewrite goal whose head is f/0.

Amount of Evaluation

Notice that once f(X) is rewritten to g(X), even though there exists a rule in the database that would cause further rewriting, no more rewriting will occur unless the term is surrounded by a monadic rewrite operator, or the term is passed to the evaluator again. Though it is possilbe to not have this scheme, the other choice – complete rewriting – causes many difficulties. In particular, it is very hard to pass terms around, since many quotes are required to prevent the evaluator from rewriting everything. Also, delayed evaluation is harder to implement.

Order of Evaluation

Two choices of order of evaluation were considered. The first was normal order evaluation (top down rewriting). In this method, the goal =>f(g(X)) matchs against a clause f(X) => whatever. This had the advantage of being simple to implement and very powerful, since large parts of terms can be rewritten at once.

The second possibility, the one finally chosen, was bottom up. This corresponds to the order of evaluation in Lisp. It is relatively straightforward to program using this notation since all of one's arguments are already evaluated. In addition it is much easier to code rewrites when one has a rather narrow set of possibilities to consider for the given arguments.

The method used in the final version traced down the trees until finding an atom, variable, integer, or quoted term. In each case the term (minus the quote, if applicable) is returned as the result

of evaluation, and the interpreter then tries to back up one level and evaluate again. Thus, backtracking will cause alternatives to be tried at the lowest levels first. Failure to rewrite at any level passes upwards, causing failure in the topmost unify.

Variables and Rewrites

A significant design decision concerns unification of variables and terms. If a bound term needs to be unified to a term of the form `=>f(something)`, rewriting must complete during unification to insure proper bindings, and proper unification. However, if a variable needs to be unified with a term of the above form, it is not necessary to do anything but a normal Prolog unification of the variable to the term. This in essence is a form of delayed (or lazy) evaluation, but also has a noticeable consequence on the semantics – it decides whether identical terms will be rewritten to the same value. Delayed evaluation gives the user the ability to handle infinite data structures. As an example, consider example six presented in Chapter four, in which lists which would be infinite if evaluated are passed as arguments.

This issue will be discussed further in the next two chapters.

Summary

To keep the semantics simple and as order independent as possible, evaluation is postponed as much as possible. Quoting and the monadic rewrite are introduced to control the power of AT1. Quoting prevents rewriting in the current goal, and the monadic rewrite in a Prolog goal starts rewriting. Within a rewrite rule, a monadic rewrite continues the rewriting. Niladic functions disambiguate terms that are to be treated as atom (which rewrite to themselves) and functions of no arguments (which need corresponding rewrite rules).

Chapter 6: Interpretation

In order to verify that the language as designed was easily usable, let alone correct, an interpreter was built. Implemented using C-Prolog, the code has also run under the Dec-20 compiler for better efficiency. This chapter gives a more detailed description of the interpreter that was mentioned in Chapter 4. In particular, the efficiency, accuracy, and completeness of the interpreter are covered.

Design Decisions

The semantics given in Chapter 4 imply that the same term, bound to a variable, can be rewritten in two different locations. In an interpreter this implies the potential for storing identical large rewrite backtracking trees and the possibility for doing the same work over and over again for a deterministic rewrite. These inefficiencies could be resolved in two different manners – evaluation upon binding with a variable, or assignment to variables.

Using evaluation upon binding, the lazy evaluation possible in AT1 would disappear. This would destroy the possibility of using infinite structures. Depending on the program, it could also introduce more delays than it saved – in the evaluation of arguments that were never used.

Another, more attractive, alternative is to use assignment to prevent repeated rewritting. Whenever a term bound to a variable would be rewritten, the newly rewritten term would be assigned to the variable. Unfortunately, this relies on an assignment predicate in Prolog. Currently no such predicates exist, though the author is currently working on this problem.

Due to these drawback, in the final version of the AT1 interpreter, neither of these choices was chosen. Instead, the semantics is left as described in chapter 4.

Accuracy and Completeness

The interpreter shown in Appendix B has several small inaccuracies, and lacks several features. Most noticeable among the lacking features is the cut operator. While there need to be no changes made to a Prolog compiler that handles cut for it to be extended to AT1, it is quite the opposite case with an interpreter. Current implementations of Prolog have no easy way for the user to guide goal execution, thus making cut all but impossible to implement.

While many of the evaluatable predicates have been hand coded into the goal `ati_prove` of the interpreter, the author has not felt impelled to be complete. Any new evaluatable predicates needed by a user program can easily be added as long as they do not pull terms apart. If this happens, code must be written as in `ati_print`. This is AT1's equivalent of Prolog's `print` predicate, and works by pulling the term apart, looking for monadic rewrites in the process.

Since the current version of the interpreter was written in Prolog, niladic functions are not implemented. Too much hacking of the Prolog reader would be required.

Efficiency

When run under C-Prolog, or, Dec-20 Prolog without the compiler, the AT1 interpreter is exceedingly inefficient. Running the maze example on a 2x3 maze takes approximately five megabytes of data space and 6.5 minutes of CPU time on a SUN. Larger samples have not been run due to the large amounts of memory and CPU time that they would use. These times must be looked upon

as the result of using two levels of interpretation – one by Prolog interpreting the AT1 interpreter, and also, the AT1 interpreter interpreting the AT1 program.

If the Dec-20 compiler is used, execution times fall drastically. Using the maze generator sample, execution times fell from around 3 minutes using the Dec-20 interpreter to around 30 seconds using the compiler. Storage is also drastically reduced. Unfortunately, the (Version KL) Dec-20's limited address space precludes larger interpreted examples. The limited address space often caused the maze generator to sometimes run out of room on even a two by three maze. The random number generator made accurate timing statistics hard to gather.

In order to get the code in Appendix B running under the Dec-20 compiler, the clauses `ati_prove(is(--,--))` must be changed to call an interpreted procedure.

Chapter 7: Compilation

Below we summarize Warren's runtime model for compiled Prolog, referring the reader to the original paper for more details. For clarity, we have simplified his treatment by deleting certain optimizations. After this introduction, the extensions necessary to compile AT1 are presented. However, first we discuss the bindings of variables to terms which include monadic rewrites.

Unlike the interpreter, it is relatively easy in the compiler design to choose either evaluation at the time of unification to a variable, or the delayed unification in case of need. This is because the compiler can generate code that will move the appropriate pointers around in either case. Due to the author's preference for being able to handle infinite terms the second choice was chosen. This also has the advantage of eliminating clause ordering dependencies (within a goal) – though argument ordering can still affect the outcome.

The Abstract Machine

Compilation translates Prolog source code into instructions for a virtual machine. These instructions are commonly implemented through small sequences of assembly language for a target machine. The virtual machine partitions its state into a number of storage areas. The first is a ***heap*** which holds atoms, constructed terms, and variables which outlast the invocation of a particular Prolog procedure. The second area is called the ***local stack***, which holds ***environments***, activation records for a procedure call, and ***choice points***, which hold information necessary for backtracking. A third area is the ***trail***, which is used to undo variable bindings upon backtracking. A fourth area is the ***code area*** which is used to hold the database. A fifth area is a set of ***argument or local registers***, are used for parameter passing and unification. The central part of the execution strategy is efficiently manipulating these registers. Beside stacking control information, the principal task of an environment is to save these registers when necessary. Finally, there is a set of machine state registers holding pointers into the above areas as well as machine state to allow execution.

Instructions

Warren classifies the instructions into four groups. They are the ***procedural*** instructions for control flow, the ***unify*** instructions for unification and copying, the ***get*** instructions for matching against procedure heads, and the ***put*** instructions for setting up arguments in subgoals.

The procedural instructions are: `allocate`, `deallocate`, `call`, `proceed`, `try_me_else`, `retry_me_else`, `trust_me_else`, `fail`, `execute`. The `allocate` and `deallocate` instruction manage space on the local stack. The `call` and `proceed` instructions do procedure linkage. The `try_me_else`, `retry_me_else`, and `trust_me_else`, instructions manage backtracking information within a procedure at the clause level. `Fail` causes backtracking. `Execute` is a simple unconditional transfer.

The get instructions are responsible for reading the argument registers. They are: `get_variable`, `get_value`, `get_constant`, `get_structure`. These are used to match the top level arguments of the head of a clause against a goal. `Get_variable` matches with an unbound variable. `Get_value` matches with a bound variable. `Get_constant` matches with an atom. `Get_structure` sets up possible copying by the unify instructions.

The unify instructions may read the argument registers or write the "local" registers (which in fact share coincident locations) when constructing subterms for a call. They are: `unify_variable`, `unify_value`, `unify_constant`, and `unify_local_value`. Each of the unify instructions either match terms in read mode or construct terms in write mode (to implement the copying strategy). `Unify_variable` and `unify_value` match with unbound and bound variables. `Unify_constant` matches against an atom. Finally, `unify_local_value` matches against a term which may be constructed locally.

The put instructions load the argument register in preparation for a procedure call. They include: `put_variable`, `put_value`, `put_unsafe_value`, `put_constant`, and `put_structure`. The `put_variable` and `put_value` instructions are used to send unbound or bound variables to the callee. The `put_unsafe_value` is a version of `put_value` needed to effect tail recursion.

Compiling Rewrite Rules

One of the more interesting features of AT1 surfaces when we consider compilation. We present a simple extension of Warren's strategy to include rewriting.

The only modification to the virtual machine is the addition of a new register, *the value register*, which points to where an evaluated result should be returned. There are no new instructions. However, we modify or extend a few of the instructions. An evaluator routine is also included. The evaluator when passed a term to evaluate, simply sets up arguments to a given rewrite predicate and calls it. In most cases, evaluation can proceed in compiled code without any invocation of the evaluator. However, there are certain cases when it is impossible to avoid interpretation. The evaluator is the vestige of an interpreter to handle those cases.

We modify the procedural instructions (`call` and `execute`) so that they may set up the value register. For example, the instruction `call p,A2` calls the procedure p and sets the value register to point to the second argument register. The procedure returns its value in argument register 2. Allocate and deallocate now must save and restore the value register.

Most of the modification occurs in the instructions used for unification. While `get_variable` instruction is unchanged, The `get_value` instruction must check to see whether the dereferenced value is an unbound variable. If so, the variable is simply bound to the argument. If the variable is bound to a structure, then when the argument contains a term whose principal functor is the monadic rewrite symbol, the evaluator is called. The other get instructions operate in the same manner: when a monadic rewrite symbol is encountered the evaluator is invoked.

The unify instructions are unchanged in the write mode. In the read mode, they operate in the same way, again, except when a monadic rewrite symbol is encountered. In these cases, the evaluator is also invoked.

The put instructions are unchanged.

Finally an in core evaluator (interpreter) must be present, for exactly the same reason that compiled Lisp code must have an interpreter present – the user may construct new function either in the database, or inside of a variable.

We now give some examples of compiled code. We mention again that many of Warren's optimizations have been deleted for clarity. They are still applicable to AT1.

First the append example,

```
%           append([],L) => L.
%           append([X|L1],L2) => [X|=>append(L1,L2)].

append/2:
   try_me_else C2
   get_const [],A1      % append([],
   get_val @val,A2      %          L) => L,  where @val indirects through the
                        %                    value register.

C2:
   trust_me_else fail
   get_struct '.'/2,A1  % append([    % '.' is the principal functor of a list.
   unify_var  X4         %        X
   unify_var  A1         %         |L1],L2) =>
   put_struct '.'/2,@val %                    [
   unify_val  X4         %                     X
   unify_var  X3         %                      |
   execute append/2,X3   %                       =>append(L1,L2)
```

Those familiar with Warren's paper will note the striking similarities between this code and his example for the Prolog append/3.

```
% g :- X = =>(1+Y), Y=1, is_two(X).
% is_two(2).

g/0:
        trust_me_else fail
        allocate
        put_var Y1,A1        % X
        put_struct +/2,X3    %       +(
        unify_const 1        %         1,
        unify_var Y2         %           Y)
        put_struct =>,A2     %     =>
        unify_var X3         %
        call =/2             %   =
        put_value Y2,A1      %               , Y
        put_const 1,A2       %                   1
        call =/2             %                  =
        put_value Y1,A1      %                    ,       (X)
        execute is_two/1     %                      is_two    .

is_two/1:
        trust_me_else fail
        get_const 2,A1       % invoke evaluator if need be  (it does!)
        proceed
```

Above, the term bound to X with a principal functor the monadic rewrite symbol is evaluated by the get_const instruction in the procedure `is_two`. The evaluator is invoked here because the rewrite expression is unified with a nonvariable for the first time in our compiled examples.

In the next example variables that are used in the rewriting are bound in proving the Prolog conditions.

```
% del_mul(N,L) => =>del_mul(N,append(H,TP)) :- append(H,[HP|TP],L), divides(N,HP).
% del_mul(N,L) => L.

del_mul/2:
        try_me_else C1
        allocate
        get_var Y1,A1     % del_mul(N,
        get_var A3,A2     %           L) =>
        put_var Y2,A1     %                          :- append(H,
        put_struct '.'/2,A2 %                                  [
        unify_var Y3      %                                   HP|
        unify_var Y4      %                                      TP],L),
        call append/3     %
        put_val Y1,A1     %                             divides(N,
        put_val Y3,A2     %                                     HP),
        call divides/2
        put_val Y2,A1     %                    (=>)    ,append(H,
        put_val Y4,A2     %                                    TP)
        call append/2,A2
        put_unsafe_val Y1,A1 %              del_mul(N     ( see just above )
        deallocate        %  get value register back
        execute del_mul/2,@val

C1:
        trust_me_else fail
        get_var @val,A2  % del_mul(N,L) => L.
        proceed
```

Chapter 8: Conclusions and Further Work

Through the explicit use of the monadic rewrite symbol combined with quotation, functional constructs form what appears to be a clean extension to Prolog. In addition this extension allows for efficient implementation via compilation. A Prolog compiler could easily be modified to compile AT1.

One particularly interesting extension to our work is to compile AT1 for lazy evaluation along the lines of Johnsson. We note that a certain amount of lazy evaluation is currently expressible within AT1. For example, through clever use of quotation and the monadic rewrite symbol the user can prevent complete evaluation when it is not wanted:

```
append([H|T],L) => #[H|=>append(T,L)].
```

will cause a partial evaluation of the original append, which may be completed through the user invoking evaluation.

We are interested in extending the above results to other data structures, in particular, arrays and directed graphs.

While the use of the Prolog tracing package is extremely useful for debugging AT1 programs, a native mode AT1 debugger would speed the debugging process. Some hybrid of the Prolog debugger and the fancier Lisp debuggers would be an extremely powerful combination.

The author feels that some research needs to be done in allowing some form of assignment to Prolog. Also, more work similar to that done by Goguen's group is needed.

The author would like to give many thanks to Jim Kajiya, who pointed the author along this path of research and is extremely interesting to work with. Howard Derby provided many hours of constructive (and fun) arguing, and Keith Hughes shed much light on how to compile Prolog.

Chapter 9: References

Apt, K., and van Emden, M.H. "Contributions to the Theory of Logic Programming" *J. ACM* 29,3 (July 1982) 841-862.

Barbuti, R., Bellia, M., Levi, G., and Martelli, M. "On the integration of logic programming and functional programming" *Proc. 1984 International Symposium on Logic Programming*, Atlantic City, 1984. 160-166.

Burstall, R.M., and Goguen, J.A. "Putting Theories Together to make Specifications", *Proceedings of the Fifth IJCAI*, 1979. pp 1045-1058.

Carlsson, M. "On Implementing Prolog In Functional Programming" *Proc. 1984 International Symposium on Logic Programming*, Atlantic City, 1984. 259-264.

Colmerarauer, A., Kanoui, H., Pasero R., and Roussel, P. "Un Système de Communication Homme-Machine en Français" *Groupe d'Intelligence Artificielle*, U.E.R. de Luminy, Université Aix-Marseille, Luminy, 1972.

van Emden,M.H., and Kowalski, R.A. "The Semantics of Predicate Logic as a Programming Language" *J. ACM* 23,4 (1976) 733-742.

Futatsugi, D., Goguen, J.A., Jouannaud, J.-P., and Meseguer, J. "Principles of OBJ2".

Goguen, J.A, and Meseguer, J. "Equality, Types, Modules and Generics for Logic Programming" *Proc. Second International Logic Programming Conference*, Uppsala University, July 1984. 115-125.

Johnsson, T. "Efficient Compilation of Lazy Evaluation" *Proc. ACM SIGPLAN 84 Symposium on Compiler Construction*. SIGPLAN Notices 19,6 June 1984. 58-69.

Pereira, F. *ed*, Warren, D., Bowen, D., Byrd, L., and Pereira, L. *C-Prolog User's Manual*, March, 1984.

Pereira, L.M., Byrd, L. Pereira, F.C.N., and Warren, D.H.D. *Users Guide to DECSystem-10 PROLOG*, DAI Occasional Paper 15, Department of Artificial Intelligence, University of Edinburgh, Scotland, September 1978.

Robinson, J.A., and Sibert, E.E. "LOGLISP: Motivations, Design, and Implementation" *Logic Programming*, Clark, K.L, and Tarnlünd, S.A. (eds.), Academic Press, 1982. 299-314.

Robinson, J.A. "A Machine-Oriented Logic Based on the Resolution Principle" *J. ACM* 12,1 (Jan 1965) 23-41.

Subrahmanyam, P.A., and You, J.H. "Pattern Driven Lazy Reduction: a Unifying Evaluation Mechanism for Functional and Logic Programs" *Proc. Eleventh ACM Symposium on Principles of Programming Languages*, 1984, 228-234.

Subrahmanyam, P.A., and You, J.H. "Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programing" *Proc. 1984 International Symposium on Logic Programming*, Atlantic City, 1984. 259-264.

Tamaki, H. "Semantics of Logic Programming with a Reducibility Predicate" *Proc. 1984 International Symposium on Logic Programming*, Atlantic City, 1984. 259-264.

Tick, E. Warren, D.H.D. "Towards a Pipelined Prolog Processor" *Proc. 1984 International Symposium on Logic Programming*, Atlantic City, 1984. 29-40.

Warren, D.H.D. "An Abstract Prolog Instruction Set" *Technical Note 309, Artificial Intelligence Center*, SRI International, October 1983

Warren, D.H.D. "An Improved Prolog Implementation which Optimizes Tail Recursion" *Research Paper* 156, Dept. of Artificial Intelligence, University of Edingburgh, 1980.

Appendix A: A Large Sample Program

```
/************************************************************************ */
/*                                                                        */
/*                      Maze Generator Example                            */
/*                                                                        */
/*  This AT1 program generates mazes of any rectangular size using a random */
/*  building process.  It is meant as a compicated test case / example    */
/*  program for the AT1 simulator.                                        */
/*  The algorithm works by building in lists the "blocks" that are to be  */
/*  added to an empty hull in order to make a maze.  There are (H-1)*(W-1) */
/*  of these blocks to add to get a maze w/o loops, and therefore only one */
/*  possible  path going from entrance to exit.  The below picture shows  */
/*  the numbering scheme used to represent the squares.  Blocks are       */
/*  represented as a list consisting of the two squares that they seperate. */
/*  For example, in the  below picture, the blocks list would be:         */
/*  [  [[1,1],[1,2]] ,                                                    */
/*     [[1,3],[2,3]] , [[1,4],[2,4]] ,                                    */
/*     [[2,5],[2,6]] ,                                                    */
/*     [[2,3],[3,3]] , [[2,4],[3,4]] , ..... ]                           */
/*                                                                        */
/*      Column #   1  2           W                                      */
/*                                                                        */
/*                 |  |           |                                      */
/*                 V  V           V                                      */
/*                                                                        */
/*                +--+--+--+--+--+--+                                     */
/*     row 1 -->     |              |                                     */
/*                +  +  +--+--+--+  +                                     */
/*     row 2 --> |              |  |                                     */
/*                +  +  +--+--+  +  +                                     */
/*                |  |     |  |  |  |                                     */
/*                +  +--+--+  +  +  +                                     */
/*     row H --> |           |  |                                        */
/*                +--+--+--+--+--+--+                                     */
/*                                                                        */
/*                                                                        */
/*                                                                        */
/* ********************************************************************** */
```

```
/* Define some standard sizes for easy typing.  Note niladic functions */
/* which are not yet handled by the interpreter! */
% smm() => =>make_maze(2,3).
% mm() => =>make_maze(3,4).

/* make_maze(Heigth, Width) -- main goal */
make_maze(H,W) =>  =>print_maze(H, W, add_segs(H,W) ) .

/* add_segs -- set up how many segments should be added in to a null maze */
/* add_segs_a -- do the adding of a randomly picked segment, check to
                 make sure the path is still possible */
add_segs(H,W) => =>add_segs_a( (H-1)*(W-1), [], H, W, all_segs(H,W)).

add_segs_a(0,L,H,W,R) => L.
add_segs_a(N, L, H, W, R) => =>add_segs_a( N-1, [New|L], H, W, Rp)
                 :- N > 0, rand_pr_pts(H,W,New),
                       append(X,[New|Y],R),
                       Rp = (=>append(X,Y)),
                       ok(New, Rp) .

/* Still possible if there exists a connection, or if there is a path to
                 an existing connection */
ok([S,E],[]) :- fail.
ok([S,E],R) :- in([S,E],R).
ok([S,E],R) :- in([E,S],R).
ok([S,E],R) :- in([E,X],R), ok([X,S],=>diff1(R,[[E,X]])).
ok([S,E],R) :- in([X,E],R), ok([X,S],=>diff1(R,[[X,E]])).

/* Find a random segment */
rand_pr_pts(H,W, [ [M,N] , [Mp,Np] ]) :- repeat,        % repeat till valid
                 ranunif(2,D),
                 ranunif(H,Pm), M is Pm+1, Mp is M+D, Mp =< H,
                 ranunif(W,Pn), N is Pn+1, Np is N+(1-D), Np =< W.

/* Find all possible seg.s: */
all_segs(H,W) =>  =>all_segs_a(1,1,H,W).
all_segs_a(H,W,H,W)=> [].
all_segs_a(M,N,H,W)=> =>[[[M,N],[M,N+1]],[[M,N],[M+1,N]]|all_segs_a(M,N+1,H,W)]
                                 :- M < H , N < W.
all_segs_a(M,N,H,W)=> =>[                [[M,N],[M+1,N]]|all_segs_a(M+1,1,H,W)]
                                 :- M < H , N = W.
all_segs_a(M,N,H,W)=> =>[[[M,N],[M,N+1]]                 |all_segs_a(M,N+1,H,W)]
                                 :- M = H , N < W.
```

```
/* Routines to print the completed maze */
print_maze(H,W,L) =>   =>
    printer( append(border(W),append(print_maze_a(0,1,0,H,W,L),border(W))) ) .
border(0) => ['+',nl].
border(N) => => ['+','-','-'|border(N-1)].
print_maze_a(0,H,W,H,W,L) => [' ',nl].
% end of line rules
print_maze_a(0,X,W,H,W,L) => ['|',nl | =>print_maze_a(1,X,0,H,W,L)].
print_maze_a(1,X,W,H,W,L) => ['+',nl | =>print_maze_a(0,X+1,0,H,W,L)] :- X < H.
% horizontal rules
print_maze_a(0,X,Y,H,W,L) => ['|',' ',' '| =>print_maze_a(0,X,Y+1,H,W,L)] :-
                ( (Y = 0 , X > 1) ; (Y > 0 , in(=>[[X,Y],[X,Y+1]],L) ) ) .
print_maze_a(0,X,Y,H,W,L) => [' ',' ',' '| =>print_maze_a(0,X,Y+1,H,W,L)].
%       Following NOT needed because of the ecut in the rule above:
%                ( (Y = 0 , X = 1) ; (Y > 0 , notin(=>[[X,Y],[X,Y+1]],L) ) ).
% vertical divider (which run horizontal) rules
print_maze_a(1,X,Y,H,W,L) => ['+','-','-'| =>print_maze_a(1,X,Y+1,H,W,L)] :-
              in(=>[[X,Y+1],[X+1,Y+1]],L) .
print_maze_a(1,X,Y,H,W,L) => ['+',' ',' '| =>print_maze_a(1,X,Y+1,H,W,L)].
%       Following NOT needed because of the ecut in the rule above:
%                notin(=>[[X,Y+1],[X+1,Y+1]],L).
printer([nl|T]) => =>printer(T) :- nl.
printer([H|T]) => =>printer(T) :- print(H).
printer([]) => 'good_luck!'.        % implied cut in true
```

```
/* Support Routines */

union(M, []) => M.
union(M, [H|T])  =>  =>union(M, T) :- in(H,M).
union(M, [H|T])  =>  =>union([H|M], T) :- notin(H,M).

diff(M, []) => M.
diff(M, [H|T])  =>  =>diff(append(A,B), T) :- append(A, [H|B], M).
diff(M, [H|T])  =>  =>diff(M, T).

% This is not right.  There should be cuts to support it!
% diff1 assumes there is only one instance in the list of the item to be removed
diff1(M, []) => M.
diff1(M, [H|T])  =>  =>append(A,B) :- append(A, [H|B], M).
diff1(M, [H|T])  =>  =>diff(M, T).

in(E, [E|_]).
in(E, [_|L]) :- in(E,L).

notin(E, []).
notin(E, [H|T]) :-   E \== H  ,  notin(E,T).

append([],L,L) .
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3) .

append(L,[]).                                    % For efficency
append([],B) => B.
append([X|Y],B) => [X|=>append(Y,B)].
```

```
% From R.A. O'Keefe 's Prolog Library
/* ***************  CProlog Version */
'$rstate'(27134, 9213, 17773).
ranunif(L,V) :-
        retract('$rstate'(A0,B0,C0)),
        A1 is (A0*171) mod 30269,
        B1 is (A0*172) mod 30307,
        C1 is (C0*170) mod 30323,
        asserta('$rstate'(A1,B1,C1)),
        T is (A1/30269.0) + (B1/30307.0) + (C1/30323.0),
        R is T-floor(T),
        V is floor(L*R).
/* */

% DEC-20 Cheapie Code:

% ranunif(L,V) :- statistics(runtime,[SS,SP]), V is SP mod L.

% Oh, and lest we forget how to do basic arithmetic:

   X+Y => Z                :- integer(X), integer(Y), Z is X+Y.
   X-Y => Z                :- integer(X), integer(Y), Z is X-Y.
   X*Y => Z                :- integer(X), integer(Y), Z is X*Y.

/* That's it folks! */
```

Appendix B: The Interpreter

```
/* AT1 -- November 1984 Edition */

/* Copyright    November, 1984,  Michael O. Newton */
/*                               Caltech 256-80  */
/*                               Pasadena, CA   91125 */

/* All Rights Reserved.  */
/* May not be used for commercial advantage */

/* Declare  ati public to the 20 compiler. */
/* Uses a slightly hacked C-Prolog which ignores public/1 and mode/1!
:-public ati/0.

/* Declare Operator Prescedences:  Dyadic rewrite, */
/* Single '=>' for invocation of rewrite rules, & the ?X = print(=>X) abrev. */
/* the '?' abbreviation */
/* and, The quasi-quote -- Note high prescedence: */
:-op(1150,yfx,=>).
:-op(950,fy,=>).
:-op(950,fy,?).
:-op(970,fy,#).

/* Mode declarations for the 20 Compiler */
:-mode ati_process(+).
:-mode ati_subprocess(+,+).
:-mode ati_prove(+).
:-mode ati_unify(?,?).
:-mode ati_eval(?,-).
:-mode ati_ceval(?,-).
:-mode ati_varsof(?,-).
:-mode ati_append(?,?,-).
:-mode ati_print(?).
:-mode ati_p(?).
:-mode ati_pl(+).
:-mode ati_patom(+).
:-mode ati_pclist(+).
:-mode ati_putchar(+).
```

```
/* The main 'face' of the interpreter */
ati  :- prompt(_,'<  '),                % Prompt of the month club
        nl,                             % show it to the user
        read(Goal),                     % get the users desire
        print('Goal is: '),             % for debugging
        print(Goal).                    % for debugging
        nl,                             % for debugging
        ati_process(Goal),
        !,
        ati.                            % and loop.
ati.                                    % For a clean looking ending
                                        % see ati_process(end_of_file).

% Take care of the case when the user types a list --
% which means to consult files for new input.
ati_process([]) :- !.
ati_process([H|_]) :- var(H), !,
        print('Cannot consult a variable'),nl,!,fail.
ati_process([H|T]) :- !, ati_consult(H), ati_process(T), see([user]).
%
% Take care of end_of_file cleanly:
%
ati_process(end_of_file) :- !, fail.
%
% Now, normal processing of user input:
%
ati_process( ?Goal ) :- !, ati_process( (print(=>Goal)) ), !.
ati_process(X) :- var(X), !, print('Statement is a variable'), nl.
ati_process(Goal) :-
        ati_varsof(Goal,L),             % extract a list of the goals vars.
        ati_prove(Goal),                % try to prove it
        ati_subprocess(Goal,L).         % make ';' or More work by putting the
                                        % cut into a seperate predicate
ati_process(Goal) :-
        nl,
        print(no),                      % we failed
        nl,
        print('Debug: '),
        print(Goal),                    % for debugging
        nl.

% Print answer and Determine if the user wants us to try another solution
ati_subprocess(Goal,L) :-
        nl,
        print(yes),nl,                  % we did it
        ati_print(L),                   % and so print the vars
        get0(More),                     % try more solutions
        ( (More = nl) ;
          ( !,                          % a well placed cut, so that when we:
            \+ ([More] = ";")           % if no, we quit
        ) ).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                        The Extended Prolog Prover                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The basic goals:

ati_prove(true).
ati_prove( (A,B) ) :- !, ati_prove(A), ati_prove(B).
ati_prove( =>A ) :- !, ati_eval(A,X), ati_prove(X).

/* Specially defined system predicates -- Because unification is so
   different, we have to do these by hand in the interpreter.  This
   would go away in a real system.  The below list is incomplete */

ati_prove( (P;Q) ) :- ati_prove(P); ati_prove(Q).
ati_prove( (P;Q) ) :- !, fail.
ati_prove( (P -> Q ; R) ) :- ati_prove(P), !, ati_prove(Q).
ati_prove( (P -> Q ; R) ) :- !, ati_prove(R).
ati_prove( (P -> Q) ) :- ati_prove(P), !, ati_prove(Q).
ati_prove( (P -> Q) ) :- !, fail.
```

```
/* Now the really bad cases -- we have to hand code each evaluateable
   predicate so that rewrites as args will work.  This is tedious.
   The list below is not complete.  It is mainly for C-Prolog.  To use
   Dec-20 Compiler, change 'is' to call an interpreted version of is! */

ati_prove( leash(X) ) :- !, leash(X).
ati_prove( trace ) :- !, trace.          % Usefule for debugging ATi code
ati_prove( nl ) :- !, nl.                % Make a newline
ati_prove( repeat ) :- repeat.
ati_prove( =>X = Y ) :- nonvar(Y), !, ati_eval(X,Z), ati_prove( Z = Y).
ati_prove( =(X,=>Y) ) :- nonvar(X), !, ati_eval(Y,Z), ati_prove( X = Z).
ati_prove( X = Y ) :- !, X = Y.
ati_prove( is(X,=>Y) ) :- !, ati_eval(Y,Z), ati_prove( X is Z ).
ati_prove( =>X is Y) :- !, ati_eval(X,Z), ati_prove( Z is Y ).
ati_prove( X is Y) :- !, X is Y.
ati_prove( atom(=>X) )  :-  !, ati_eval(X,Y), ati_prove(atom(Y)).
ati_prove( atom(X) ) :- !, atom(X).
ati_prove( var(=>X) ) :-  !, ati_eval(X,Y), ati_prove(var(Y)).
ati_prove( var(X) ) :- !, var(X).
ati_prove( integer(=>X) ) :- !, ati_eval(X,Y), ati_prove(integer(X)).
ati_prove( integer(X) ) :- !, integer(X).
ati_prove( print(Y) ) :- !, ati_print(Y).
ati_prove( '>'(=>X,Y) ) :- !, ati_eval(X,Z), ati_prove( '>'(Z,Y) ).
ati_prove( '>'(X,=>Y) ) :- !, ati_eval(Y,Z), ati_prove( '>'(X,Z) ).
ati_prove( X > Y )  :- !, X > Y.
ati_prove( '<'(=>X,Y) ) :- !, ati_eval(X,Z), ati_prove( '<'(Z,Y) ).
ati_prove( '<'(X,=>Y) ) :- !, ati_eval(Y,Z), ati_prove( '<'(X,Z) ).
ati_prove( X < Y )  :- !, X < Y.
ati_prove( statistics(A,=>B) ) :- !, ati_eval(B,Z), statistics(A,Z).
ati_prove( statistics(=>A,B) ) :- !, ati_eval(A,Z), statistics(Z,B).
ati_prove( statistics(A,B) ) :- !, statistics(A,B).
ati_prove( =>X \== Y ) :- nonvar(Y), !, ati_eval(X,Z), ati_prove( Z \== Y).
ati_prove(  \==(X,=>Y) ) :- nonvar(X), !, ati_eval(Y,Z), ati_prove( X \== Z).
ati_prove( X \== Y ) :- !, X \== Y.
ati_prove( =>X =< Y ) :- nonvar(Y), !, ati_eval(X,Z), ati_prove( Z =< Y).
ati_prove( =<(X,=>Y) ) :- nonvar(X), !, ati_eval(Y,Z), ati_prove( X =< Z).
ati_prove( X =< Y ) :- !, X =< Y.
ati_prove( retract(=>X) ) :- !, ati_eval(X,Z), ati_prove( retract(Z) ).
ati_prove( retract(X) ) :- !, retract(X).
ati_prove( asserta(=>X) ) :- !, ati_eval(X,Z), ati_prove( asserta(Z) ).
ati_prove( asserta(X) ) :- !, asserta(X).
ati_prove( \+(X) ) :- ati_prove(X), !, fail.
ati_prove( \+(X) ).
ati_prove( not(X) ) :- ati_prove(X), !, fail.
ati_prove( not(X) ).
ati_prove( fail ) :- !, fail.
```

```
/* ***************** Normal Call to Clause ***************** */
/* If none of the above special cases succeeded, then use the AT1 interpreter
   below to satisfy the goal.  Here is where everything special happens! */
ati_prove(A) :- ati_clause(A,B), ati_prove(B).

% get a candidate clause from the database.  First make sure we are not
% trying to prove a variable.  Done here for efficency reasons.
ati_clause(A,Q) :- var(A), print("Cant prove var.  Bye"), !, fail.
ati_clause(A,Q) :-  functor(A,Functor,Arity),
                    functor(P,Functor,Arity),
                    clause(P,Q),
                    ati_unify(A,P).

% Unification modified to call eval when necessary.
% This is where the standard Prolog interpreter is modified to call evaluation
ati_unify(X,Y) :- (var(X); var(Y)), !, X=Y.        % Should be first
ati_unify(=>X, Proc) :- nonvar(Proc), !, ati_eval(X,Y), ati_unify(Y,Proc).
ati_unify(=>X, Proc) :- var(Proc), !, Proc = (=>X).
ati_unify(Proc, =>X) :- var(Proc), !, Proc = (=>X).
ati_unify(Proc, =>X) :- nonvar(Proc), !, ati_eval(X,Y), ati_unify(Y,Proc).
ati_unify([],[]) :- !.
ati_unify([H|T], [H1|T1]) :- !, ati_unify(H,H1), ati_unify(T,T1).
ati_unify(#X,#X) :- !.
ati_unify(X, Proc) :-
                X   =..  [Functor|Xargs],
                Proc =.. [Functor|Pargs],
                ati_unify(Xargs,Pargs).

/* ati_eval  and ati_ceval are the evaluator and it's conditional side-kick,
   the conditional evaluator (remember that after something has been evaluated
   it is no longer failure if it cannot continue). */
ati_eval(X,X) :- (integer(X); atom(X); var(X)), !.
ati_eval(#X,X) :- !.
ati_eval(=>X,Y) :- !, ati_eval(X,Y).
ati_eval([],[]) :- !.
ati_eval([H1|T1], [H2|T2]) :- !, ati_eval(H1,H2), ati_eval(T1,T2).
ati_eval(X,Z) :- !,
                 functor(X,Functor,Arity),
                 functor(Y,Functor,Arity),      % Make a dummy of same form
                 X =.. [Functor|Xargs],         % Take care of args first
                 ati_eval(Xargs,XXargs),
                 XX =.. [Functor|XXargs],       % Build new ars
                 clause( =>(Y,T) , Q ),         % Get an
                 ati_unify(XX,Y),               %         appropriate clause
                 ati_prove(Q),
                 ati_ceval(T,Z).                % See if more work to do?
ati_ceval(X,X) :- (integer(X); atom(X); var(X)), !.
ati_ceval(#X,#X) :- !.                              %investigate no further
ati_ceval(=>X,Z) :- !, ati_eval(X,Z).               % Yes there is.
ati_ceval([],[]) :- !.
ati_ceval([H1|T1],[H2|T2]) :- !, ati_ceval(H1,H2), ati_ceval(T1,T2).
ati_ceval(X,Y) :- !,
                X =.. [Functor|Args],
                ati_ceval(Args,Nargs),
                Y =.. [Functor|Nargs].
```

```
/* find all the variable is a statement.  May find the same variable twice.
   Could probably be much more efficently written.  */
ati_varsof( T , [T]) :- var(T), !.
ati_varsof( T , [] ) :- atomic(T) , !.
ati_varsof( [] , []) :- !.
ati_varsof( [T|U] . X ) :- !.           % cant do this another way
        ati_varsof(T,H), !,             % nor this
        ati_varsof(U,R), !,             % nor this
        ati_append(H,R,X), !.           % now combine them
ati_varsof( T, L) :- !,
        T =.. [F|Args], !,
        ati_varsof(Args,L), !.

ati_append([],X,X).
ati_append([H|T],X,[H|F]) :- ati_append(T,X,F).
```

```
/* ************************************************************** */
/* A very short addition of the prolog predicates print and putatom */
/* Inserted since we have to unify against them with our new unify. */
/* This is faked in the current scheme so that these can be used.   */
/* ************************************************************** */

ati_print(=>X) :- !, ati_eval(X,Y), ati_p(Y).
ati_print(X) :- ati_p(X).

ati_p(X) :-  atom(X), !, ati_patom(X).
ati_p(X) :-  integer(X), !, ati_patom(X).
ati_p(X) :-  var(X), !, ati_patom('?').
ati_p( .(X,[]) ) :- !, ati_putchar('['), ati_p(X), ati_putchar(']').
ati_p( .(X,Y) ) :- !, ati_putchar('['),
                   !, ati_p(X),
                   !, ati_putchar(','),
                   !, ati_pl(Y),
                   !, ati_putchar(']').
ati_p(X) :-  !, X =.. [Functor|Args],
                   !, ati_p(Functor),
                   !, ati_putchar('('),
                   !, ati_pl(Args),
                   !, ati_putchar(')').

ati_pl(=>X) :- !, ati_eval(X,Y), ati_pl(Y).
ati_pl([]) :- !.
ati_pl([H|[]]) :- !, ati_p(H).
ati_pl([H|T]) :- ati_p(H), ati_putchar(','), ati_pl(T).

ati_patom([]) :- ati_putchar('['), ati_putchar(']').
ati_patom(X) :- name(X,L), ati_pclist(L).

ati_pclist([]) :- !.
ati_pclist([H|T]) :- put(H), ati_pclist(T).

ati_putchar(H) :- name(H,[L]), put(L).
```